浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：范源颢

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3180103574

指导教师：陈文智

2020 年　30 月 30　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：单周期 CPU 设计

学生姓名：范源颢 专业：计算机科学技术 学号：3180103574

同组学生姓名：周寒靖 指导老师：陈文智

实验地点：曹光彪西楼-301 实验日期：2020 年 10 月 30 日

一、 实验目的和要求

目的：

1. 了解单周期 CPU 控制器的原理，掌握单周期 CPU 控制器的设计方法；
2. 了解数据通路的原理并且掌握设计数据通路的方法；
3. 了解单周期 CPU 的原理并且掌握设计单周期 CPU 的主要方法；
4. 掌握 CPU 程序验证的方法。

要求：

1. 设计 CPU 控制器和数据通路，组合这些基本单元形成一个单周期 CPU。
2. 用程序验证单周期 CPU，观察程序的执行情况。

二、 实验内容和原理

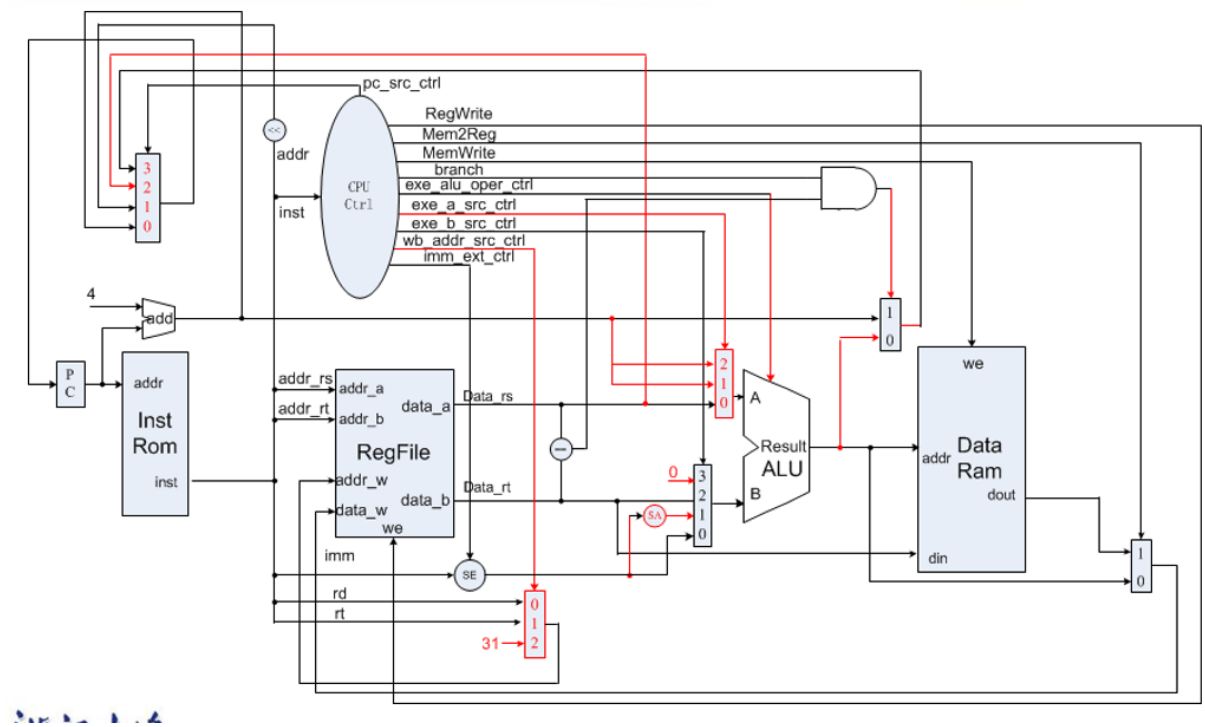
2.1 我们在本次试验中仅仅实现 16 指令的 MIPS 汇编语言。

各条语句的语法和含义如下：

16 MIPS Instructions

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations	
R-type	op	rs	rt	rd	sa	func		
add		rs	rt	rd	00000	100000	rd = rs + rt; with overflow	PC+=4
sub		rs	rt	rd	00000	100010	rd = rs - rt; with overflow	PC+=4
and		rs	rt	rd	00000	100100	rd = rs & rt;	PC+=4
or	000000	rs	rt	rd	00000	100101	rd = rs | rt;	PC+=4
sll		00000	rt	rd	sa	000000	rd = rt << sa;	PC+=4
srl		00000	rt	rd	sa	000010	rd = rt >> sa (logical);	PC+=4
slt		rs	rt	rd	00000	101010	if(rs < rt)rd = 1; else rd = 0; <(signed)	PC+=4
jr		rs	00000	00000	00000	001000		PC=rs
I-type	op	rs	rt		immediate			
addi	001000	rs	rt		imm		rt = rs + (sign_extend)imm; with overflow	PC+=4
andi	001100	rs	rt		imm		rt = rs & (zero_extend)imm;	PC+=4
ori	001101	rs	rt		imm		rt = rs | (zero_extend)imm;	PC+=4
lw	100011	rs	rt		imm		rt = memory[rs + (sign_extend)imm];	PC+=4
sw	101011	rs	rt		imm		memory[rs + (sign_extend)imm] <-- rt;	PC+=4
beq	000100	rs	rt		imm		if (rs == rt) PC+=4 + (sign_extend)imm <<2;	PC+=4
J-type	op			address				
j	000010			address			PC = (PC+4)[31..28],address<<2	
jal	000011			address			PC = (PC+4)[31..28],address<<2 ; $31 = PC+4	

2.2 实验设计的电路图如下所示：



2.3 CPU 控制器的信号有如下四条值得注意：

1. pc_src_ctrl(inst_addr):用于指定 CPU 需要执行的下一条汇编语句：
    a) Default: PC+4 //递增
    b) PC_JUMP: {inst_addr[31:28], inst_data[25:0], 2'b0} //实现 Jump 指令
    c) PC_JR: data_rs //实现 jr 指令
    d) PC_BEQ: alu_out //实现 branch 指令
2. wb_addr_src_ctrl (regw_addr): 用于指定写回寄存器的地址
    a) WB_ADDR_RD: addr_rd //写回寄存器地址是指令的 rd 字段 （Rtype 类型）
    b) WB_ADDR_RT: addr_rt //写回寄存器地址是指令的 rs 字段（lw 指令）
    c) WB_ADDR_LINK: GPR_RA //协会寄存器是 GPR 类型的值
3. exe_a_src_ctrl(opa)：ALU 操作数 a 的来源控制
    a) EXE_A_RS: data_rs //来源于指令的 RS 字段，（Rtype 类型）
    b) EXE_A_LINK: inst_addr_next //来源于 PC 寄存器中的地址，用于 Jump 指令运算下一条需要跳转的指令的地址。
    c) EXE_A_BRANCH: inst_addr_next //来源于 PC 寄存器中的地址，用于 Branch 指令运算下一条需要跳转的指令的地址。
4. exe_b_src_ctrl(opb)：ALU 操作数 b 的来源控制
    a) EXE_B_RT: data_rt //来源于指令的 RT 字段，（Rtype 类型）
    b) EXE_B_IMM: data_imm //来源于指令的立即数字段
    c) EXE_B_LINK: 32'h0 //固定为 32 位零，用于 Jump 指令的计算
    d) EXE_B_BRANCH: {data_imm[29:0], 2'b0} //修正过的 Branch 指令的立即数，用于实现 branch 指令的跳转。

2.4 我们的主要工作：

1. 处理 CPUController （我们需要补全空白的代码）
2. 处理 CPU ALU（我们需要补全空白的代码）
3. 处理 CPU 的 Register file（这一部分已经在代码框架中的 refile[addr_w] <= data_w 实现，注意我们可以使用语法 reg [31:0] regfile[1:31]实现一个包含 30 个元素，每个元素 32 位的寄存器数组。）
4. 处理 CPU 的 Instruction Memory 和 Data Memory

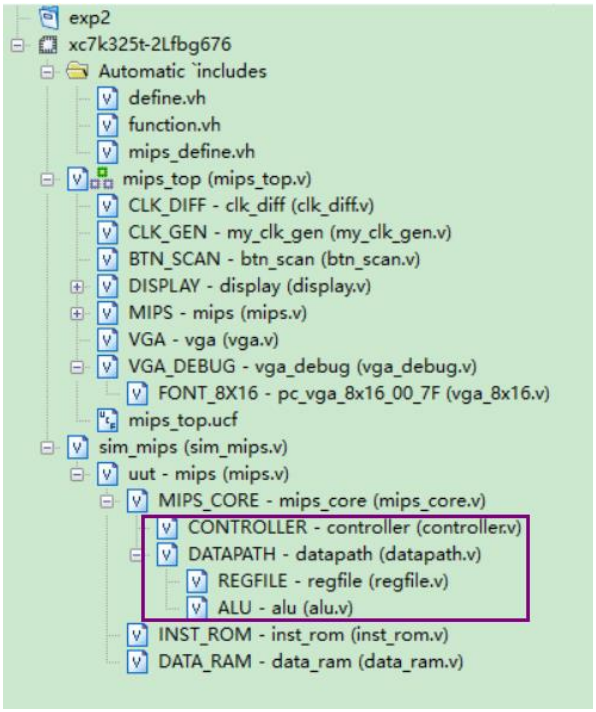
   （这一部分已经通过代码框架中的 $readmemh(“data_mem.hex”,data) 和

$readmemh(“inst_mem.hex”,data)即可实现，在文件 inst_mem.hex 中书写 16 进制机器码，并且在 data_mem.hex 中设定初值，就可以运行程序。）

5. 设计验证程序：见下文。

三、 实验过程和数据记录

实验的文件结构如下：



Automatic includes 文件夹下定义了我们可以在代码中使用的头文件；

mips_top.v 是实验的主要工程，实现了对于 MIPS 指令的模拟，clk_diff 和 clk_gen 模块用于 CPU 内置时钟的信号生成的处理，便于各个进程的同步。Btn_san 是处理输入信号的模块，时限完成的 mips_top 模块将烧录到实验室的 sword 开发板上，通过按钮的控制反映 CPU 的计算情况，信息通过 3 个模块输出，分别是 DISPLAY 模块和 VGA，VGA_DEBUG 模块，mips_top.ucf 规定了电路的引脚，使得烧录之后的程序可以嵌入开发板；

sim_mi.v 实际上是一个测试用的工程文件，用于在开发者的电脑上，用 Xilinx ISE 仿真烧录之后的输入输出过程，uut 文件指定了我们的模拟方式（通过内置时钟不断读取 inst_rom 和 data_ram 中的信息，并且把相关的信号反馈到电路图上），仿真检测的对象自然是 mips_core，这个 mips_core 核上文中 mips_top 模块中的 mips_core 是同一个文件。

我们需要完成 mips_core 中的代码补全。mips.v 文件划分为 MIPS_CORE 和 inst_rom，data_ram，完成了二者（CPU 和内存）的交互，mips_core.v 划分为 controller.v 和 datapath.v，完成了二者（控制器和数据通路）的交互，我们重点需要完成 controller.v 和 datapath.v 的代

码补全。

1. CPU Datapath：

在 CPU Datapath 中，我们可以从 inst_rom 读取了当下需要执行的指令的 32 进制机器码，同时接受了 controller 的指令。

于是，我们主要需要实现上文中四个 CPU 控制信号对于数据通路的指示：

在时钟上沿，我们需要从 inst_rom 中读取指令，为此，我们需要提供 isnt_rom 地址 inst_addr，根据上文中 CPU 控制信号，我们有：

```
always @(posedge clk) begin
        if (cpu_rst) begin
            inst_addr <= 0;
        end
        else if (cpu_en) begin
            case (pc_src_ctrl)
                PC_JUMP: inst_addr <=
{inst_addr[31:28],inst_data[25:0],2'b0};//
                PC_JR: inst_addr <= data_rs;//
                PC_BEQ: inst_addr <= rs_rt_equal ? alu_out :
inst_addr_next;//判断分支！！
                PC_BNE: inst_addr <= rs_rt_equal ? inst_addr_next :
alu_out;//判断分支！！
                default: inst_addr <= inst_addr_next;//
            endcase
        end
    end
```

之后，通过 mips_core 模块将 inst_addr 输出给 inst_rom，inst_rom 返回 16 进制机器码给 controller.v，controller.v 将机器码解读出来再返回 addr_rd，addr_rt 给 datapath.v 我们可以进一步处理之后四个 ctrller 信号：

写回信号（包括写回的地址和数据值）：

```
always @(*) begin
        regw_addr = inst_data[15:11];
        case (wb_addr_src_ctrl)
            WB_ADDR_RD: regw_addr = addr_rd;//
            WB_ADDR_RT: regw_addr = addr_rt;//
            WB_ADDR_LINK: regw_addr = GPR_RA;
        endcase
    end
```

```
always @(*) begin
        regw_data = alu_out;
        case (wb_data_src_ctrl)
            WB_DATA_ALU: regw_data = alu_out;//
            WB_DATA_MEM: regw_data = mem_din;//
        endcase
    end
```

ALU 操作符信号

```
always @(*) begin
        opa = data_rs;
        opb = data_rt;
        case (exe_a_src_ctrl)
            EXE_A_RS: opa = data_rs;//
            EXE_A_LINK: opa = inst_addr_next;//
            EXE_A_BRANCH: opa = inst_addr_next;//
        endcase
        case (exe_b_src_ctrl)
            EXE_B_RT: opb = data_rt;//
            EXE_B_IMM: opb = data_imm;//
            EXE_B_LINK: opb = 32'h0;//
            EXE_B_BRANCH: opb = {data_imm[29:0],2'b0};//
        endcase
    end
```

完成代码填空。

2. CPU Controller

当我们得到 32 位的机器码时，我们需要根据机器码各个段位的内容将相应的内容抽取出来，同时根据机器码的语义发出相应的信号，具体如下，我们采用二级译码的方式，首先判断 Rtype 指令，我们根据他们的 func 字段给出信号，以减法为例：

```
case (inst[31:26])
            INST_R: begin
                case (inst[5:0])
                    R_FUNC_JR: begin
                        pc_src = PC_JR;
                    end
                    R_FUNC_ADD: begin
                        ...;
                    end
                    R_FUNC_SUB: begin
                        exe_alu_oper = EXE_ALU_SUB;//
                        wb_addr_src = WB_ADDR_RD;//
                        wb_data_src = WB_DATA_ALU;//
```

```
                    wb_wen = 1;
                end
                R_FUNC_AND: begin
                    ...;
                end
                R_FUNC_OR: begin
                    ...;
                end
                R_FUNC_SLT: begin
                    ...;
                end
                default: begin
                    unrecognized = 1;
                end
            endcase
        end
...;
endcase
```

我们指定 ALU 的运算方法为减法，EXE_ALU_SUB，并且指定相应的写回内容，其中 exe_a_src 和 exe_b_src 已经预设为 rs 和 rt,所以这个代码块中不再做修改。

对于 JAL 和 J 指令：

```
case (inst[31:26])
        ...;
        INST_J: begin
            pc_src = PC_JUMP;//
        end
        INST_JAL: begin
            pc_src = PC_JUMP;//
            exe_a_src = EXE_A_LINK;//
            exe_b_src = EXE_B_LINK;//
            exe_alu_oper = EXE_ALU_ADD;//
            wb_addr_src = WB_ADDR_LINK;//
            wb_data_src = WB_DATA_ALU;//
            wb_wen = 1;
        end
         ...;
endcase
```

我们需要发出 pc_src 信号提醒 datapath 修改 PC 指令，jal 语句需要将当前地址存入寄存器中，于是需要额外的 ALU 操作信号和写回信号。

Branch 类型的指令：

```
case (inst[31:26])
          ...;
          INST_BEQ: begin
              pc_src = PC_BEQ;//
              exe_a_src = EXE_A_BRANCH;//
              exe_b_src = EXE_B_BRANCH;//
              exe_alu_oper = EXE_ALU_ADD;//
              imm_ext = 1;
          end
          INST_BNE: begin//add myself
              pc_src = PC_BNE;//
              exe_a_src = EXE_A_BRANCH;//
              exe_b_src = EXE_B_BRANCH;//
              exe_alu_oper = EXE_ALU_ADD;//
              imm_ext = 1;
          end
          ...;
endcase
```

注意我们在 datapath 中已经判断过了 rs 和 rt 是否相等，因此这里不需要再用 ALU 进行比较，我们需要的是，利用加法计算出需要跳转的地址号。

I 类型指令：

```
case (inst[31:26])
          ...;
          INST_ADDI: begin
              imm_ext = 1;
              exe_b_src = EXE_B_IMM;
              exe_alu_oper = EXE_ALU_ADD;
              wb_addr_src = WB_ADDR_RT;
              wb_data_src = WB_DATA_ALU;
              wb_wen = 1;
          end
          INST_ANDI: begin
              imm_ext = 0;//是 0
              exe_b_src = EXE_B_IMM;//
              exe_alu_oper = EXE_ALU_AND;//
              wb_addr_src = WB_ADDR_RT;//
              wb_data_src = WB_DATA_ALU;//
              wb_wen = 1;//
          end
          INST_ORI: begin
```

```
            imm_ext = 0;//是 0
            exe_b_src = EXE_B_IMM;
            exe_alu_oper = EXE_ALU_OR;
            wb_addr_src = WB_ADDR_RT;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
        end
        ...;
endcase
```

和 R 类型类似，注意控制 ALU 的 B 操作数是代码中的立即数，其他的写回指令和 Rtype 完全一致。注意对于 addi，我们需要的带符号扩展，所以 imm_ext 是 1，ori 和 andi 不考虑符号，使用 imm_ext= 0；

lw 和 sw:

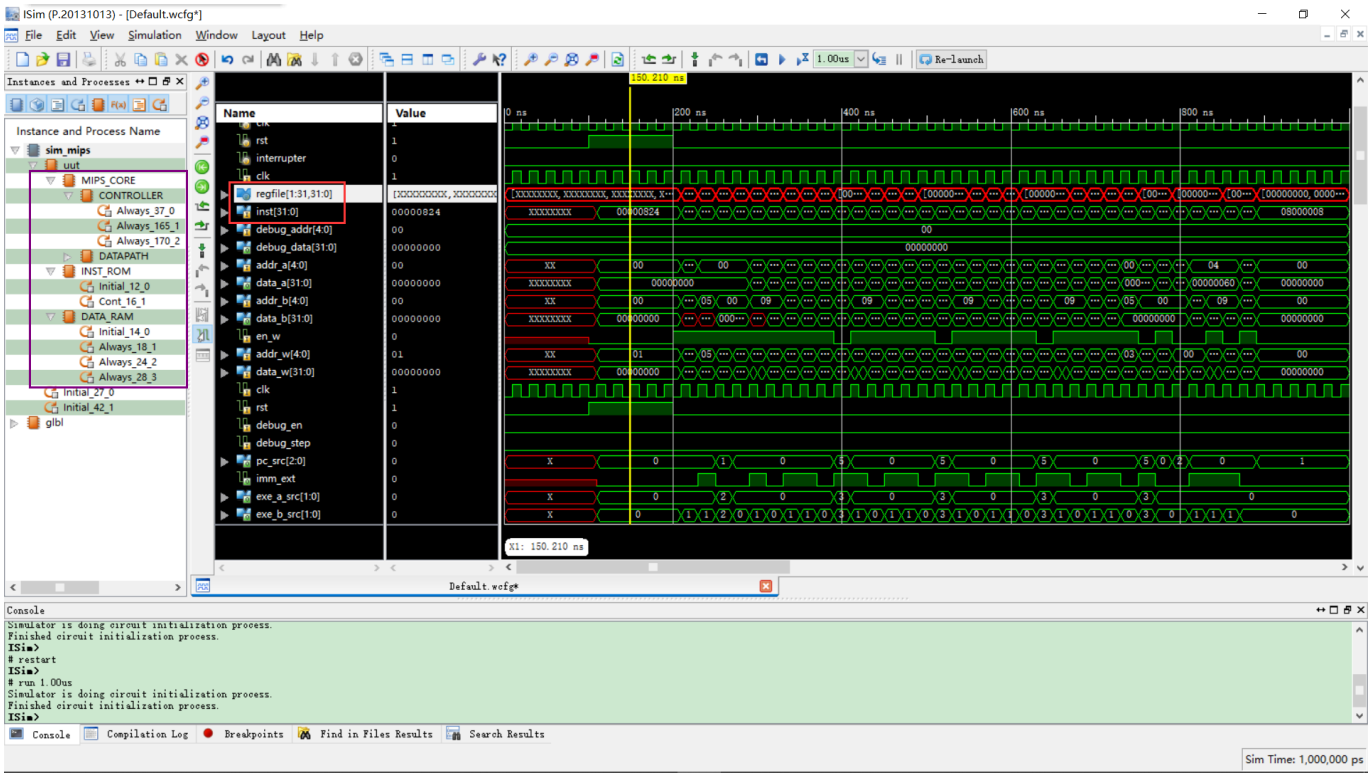
```
case (inst[31:26])
        ...;
        INST_LW: begin
            imm_ext = 1;//
            exe_b_src = EXE_B_IMM;//
            exe_alu_oper = EXE_ALU_ADD;//
            mem_ren = 1;//
            wb_addr_src = WB_ADDR_RT;//
            wb_data_src = WB_DATA_MEM;//
            wb_wen = 1;
        end
        INST_SW: begin
            imm_ext = 1;//
            exe_b_src = EXE_B_IMM;//
            exe_alu_oper = EXE_ALU_ADD;//
            mem_wen = 1;//
        end
        ...;
endcase
```

lw 和 sw 都需要 imm_ext 带符号扩展之后，设定 exe_b_src 为立即数，然后使用加法计算出需要读写的地址，因为 sw 仅仅是读出数据，所以只需要设计内存可读即可 mem_wen = 1（MEMory_Write_ENable），而 lw 还需要一步写回，所以需要设定内存可写，以及指定写回的内容和地址。

至此完成 controller 的填空，也完成了所有工程的填空。

四、 实验结果分析

仿真结果：



我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 regfile 和 inst 信号的输出，通过观察 inst 信号我们可以确定跳转的指令是否正确，通过观察 regfile 我们可以确定写回的信号是否正确。

我们测试用的程序如下：

```
00000824 main:      and $1, $0, $0      # address of data[0]
34240050            ori $4, $1, 80      # address of data[0]
20050004 call:      addi $5, $0, 4      # counter
0c00000a            jal sum             # call function
ac820000 return:    sw $2, 0($4)        # store result
8c890000            lw $9, 0($4)        # check sw
ac890004            sw $9, 4($4)        # store result again
01244022            sub $8, $9, $4      # sub: $8 <= $9 - $4
08000008 finish:    j finish            # dead loop
00000000            nop                 # done

00004020 sum:       add $8, $0, $0      # sum function entry
8c890000 loop:      lw $9, 0($4)        # load data
01094020            add $8, $8, $9      # sum
20a5ffff            addi $5, $5, -1     # counter - 1
```

```
20840004            addi $4, $4, 4      # address + 4
0005182a            slt $3, $0, $5      # finish?
1460fffa            bne $3, $0, loop    # finish?
01001025            or $2, $8, $0       # move result to $v0
03e00008            jr $ra              # return
00000000            nop                 # done
```

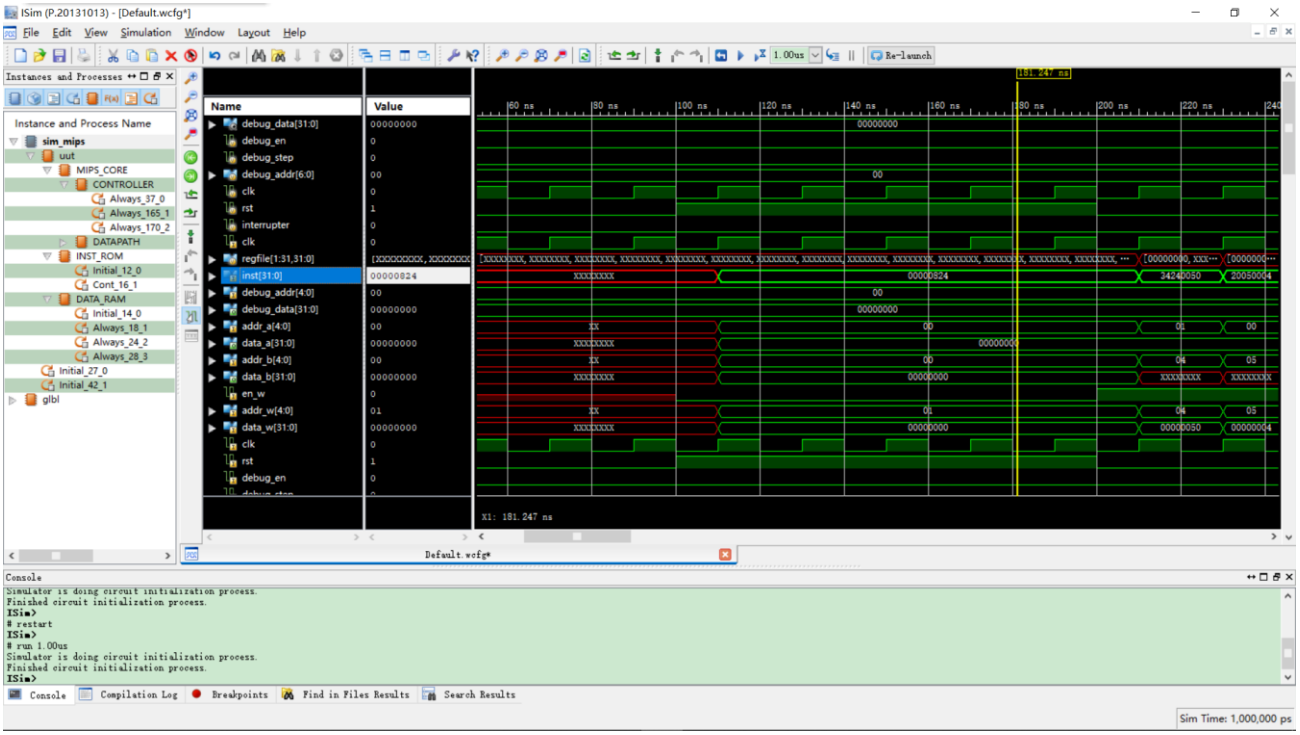
最右侧的 16 进制数是机器码，直接存放在 inst_rom.hex 中，被 CPU 读取。左侧的内容是机器码翻译之后的汇编语言，以及汇编语言的语义（用注释表示）

设定 1 号和 4 号寄存器的数据之后，我们调用函数 sum 将 80 号寄存器的值写入 9 号寄存器，注意我们同样根据 MIPS 规则将每个寄存器存入 32 位（4 字节）的值，于是 80 号相当于第 21 个字（因为 0 号是第一个字），查找 data_ram.hex 文件可以找到他的初始值是 a3，之后 sum 函数给 5 号寄存器不断减 1（5 号初值为 4），同时 4 号寄存器不断加 4（这样每次 9 号寄存器读取出来的值都是内存中下一个字的值）。8 号寄存器则不断累加 9 号寄存器的值，在 5 号小于零前不断循环，最终。将这个值返回给 2 号寄存器。此时四号寄存器已经是 0x60(16 进制的 60 就是十进制的 96， = 80 + 4 * 4)，我们将累加的结果从 2 号寄存器存入内存中的第 25 个字，之后重新读出为 9 号寄存器的值，其值为 0x258，然后我们再减去 4 号寄存器的值 0x60,得到的结果就是八号寄存器的值，应当为 0x1f8.

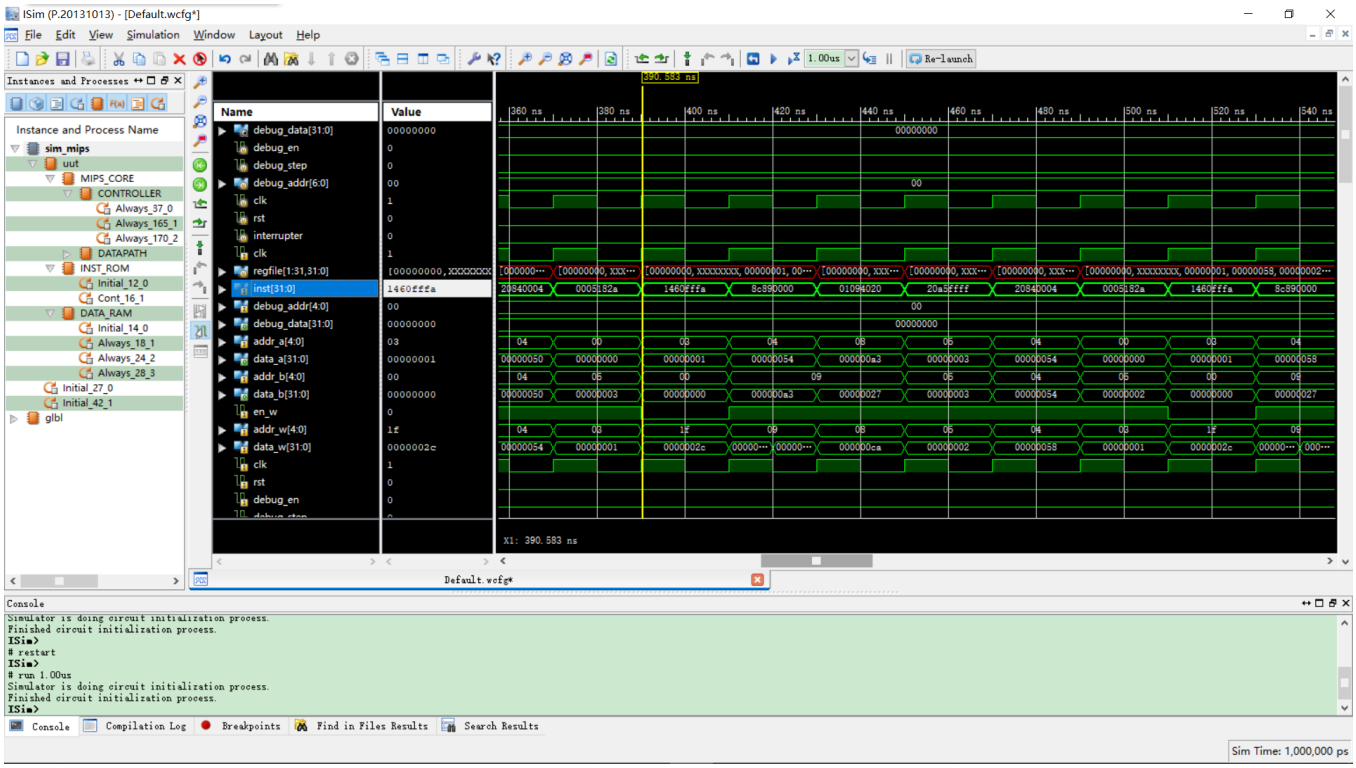
于是，假若 8 号寄存器的最终值为 0x1f8，4 号为 0x60,9 号为 0x258，我们就有很大把握这个 CPU 是正确的。

我们还可以通过 inst 寄存器的输出判断代码的执行顺序，根据汇编语言的逻辑，我们首先从 00000824 顺序执行到 0c00000a（jal 指令），之后跳转到 00004020(sum 位置)执行到 1460ffa(bne),之后跳转为 8c890000(loop 位置),反复执行 4 次，第 4 次到达 1460ffa 之后从 01001025 向下，在 03e00008（jr 指令）出跳转回主程序 ac820000(retrun 位置)最后顺序执行到底，在 0800008（finish 位置）重复跳转到自身，执行死循环。

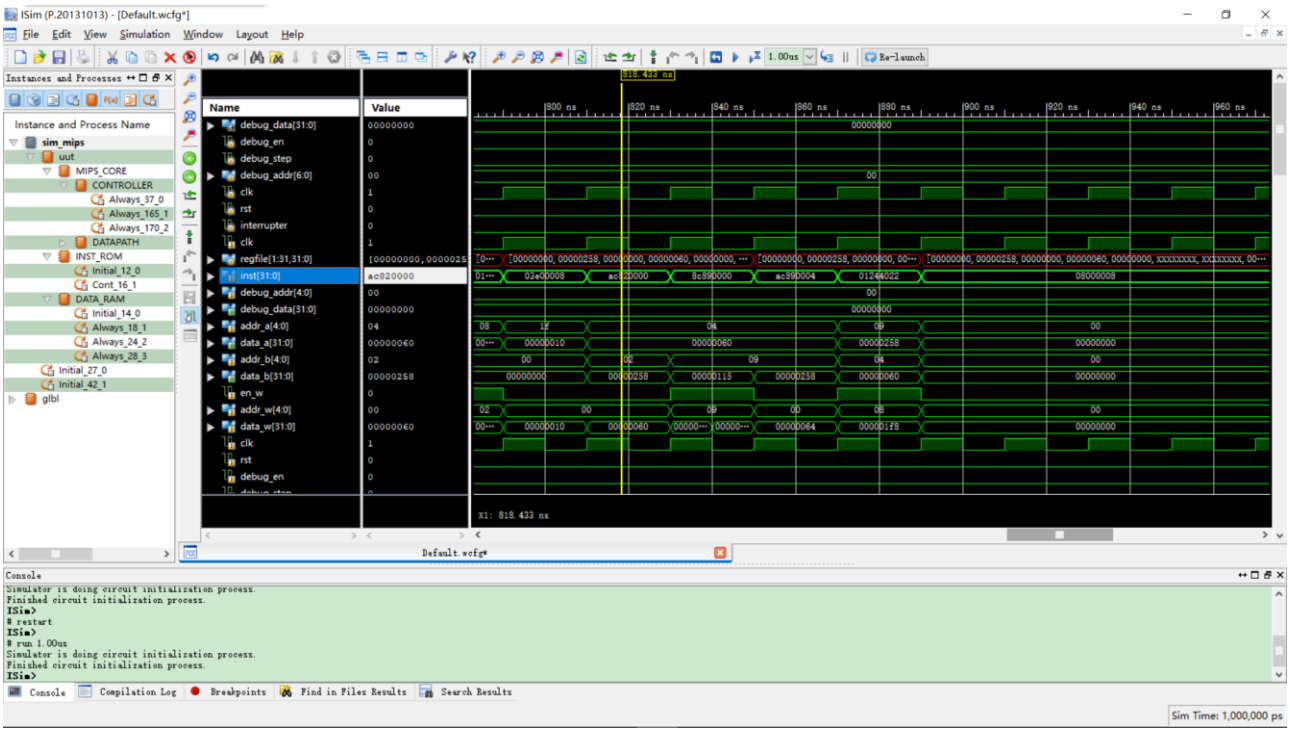
Inst 部分的波形（初始）：


ISim (P.20131013) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
Instances and Processes
Instance and Process Name
sim_mips
uut
MIPS_CORE
CONTROLLER
Always_37_0
Always_165_1
Always_170_2
DATAPATH
INST_ROM
Initial_12_0
Cont_16_1
DATA_RAM
Initial_14_0
Always_18_1
Always_24_2
Always_28_3
Initial_27_0
Initial_42_1
glbl
Name
Value
debug_data[31:0]
debug_en
debug_step
debug_addr[6:0]
clk
rst
interrupter
clk
regfile[1:31,31:0]
inst[31:0]
debug_addr[4:0]
debug_data[31:0]
addr_a[4:0]
data_a[31:0]
addr_b[4:0]
data_b[31:0]
en_w
addr_w[4:0]
data_w[31:0]
clk
rst
debug_en
Default.wcfg*
Console
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
restart
ISim>
run 1.00us
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 1,000,000 ps

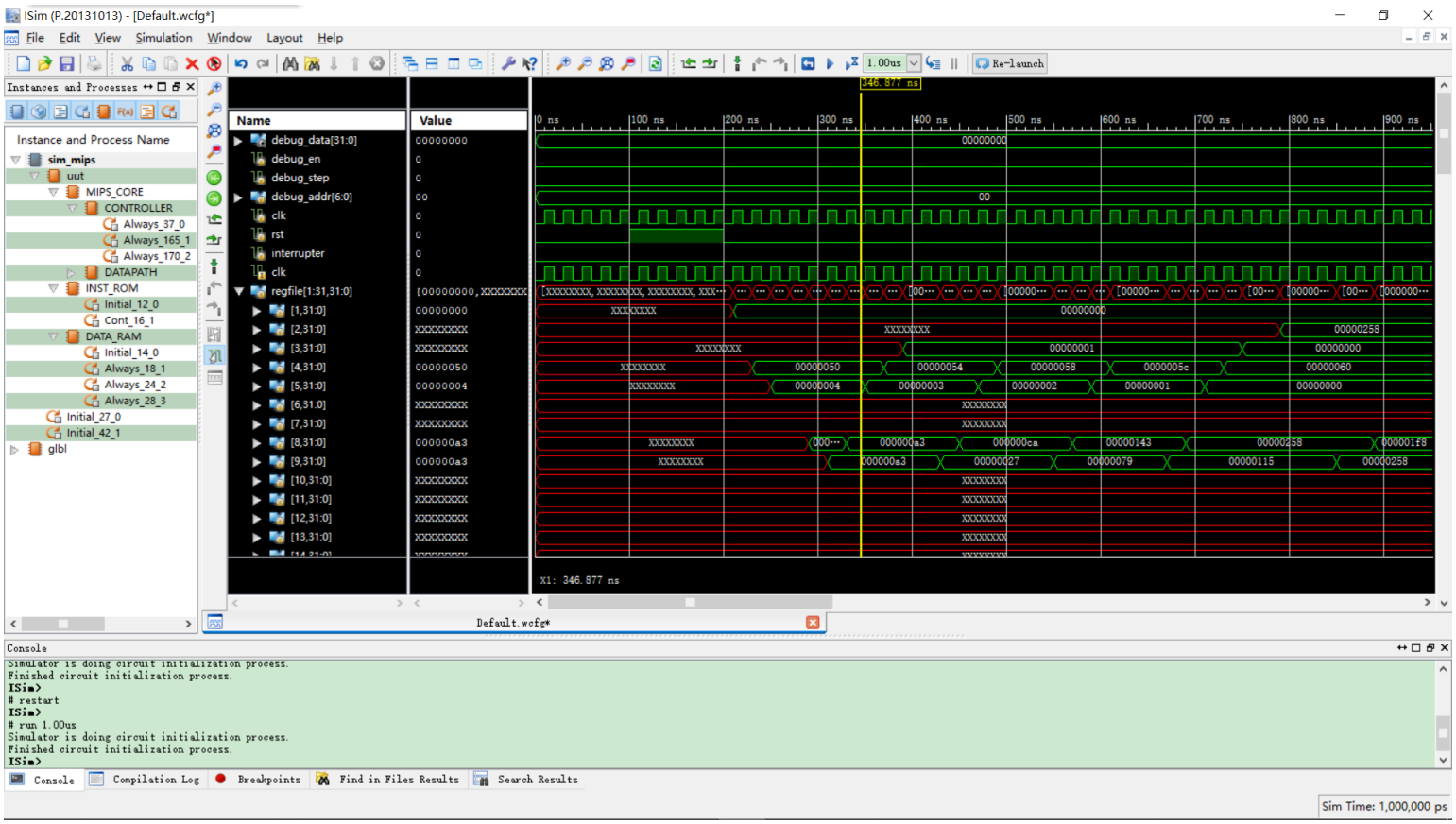
（循环部分）


ISim (P.20131013) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
Instances and Processes
Instance and Process Name
sim_mips
uut
MIPS_CORE
CONTROLLER
Always_37_0
Always_165_1
Always_170_2
DATAPATH
INST_ROM
Initial_12_0
Cont_16_1
DATA_RAM
Initial_14_0
Always_18_1
Always_24_2
Always_28_3
Initial_27_0
Initial_42_1
glbl
Name
Value
debug_data[31:0]
debug_en
debug_step
debug_addr[6:0]
clk
rst
interrupter
clk
regfile[1:31,31:0]
inst[31:0]
debug_addr[4:0]
debug_data[31:0]
addr_a[4:0]
data_a[31:0]
addr_b[4:0]
data_b[31:0]
en_w
addr_w[4:0]
data_w[31:0]
clk
rst
debug_en
Default.wcfg*
Console
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
restart
ISim>
run 1.00us
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 1,000,000 ps

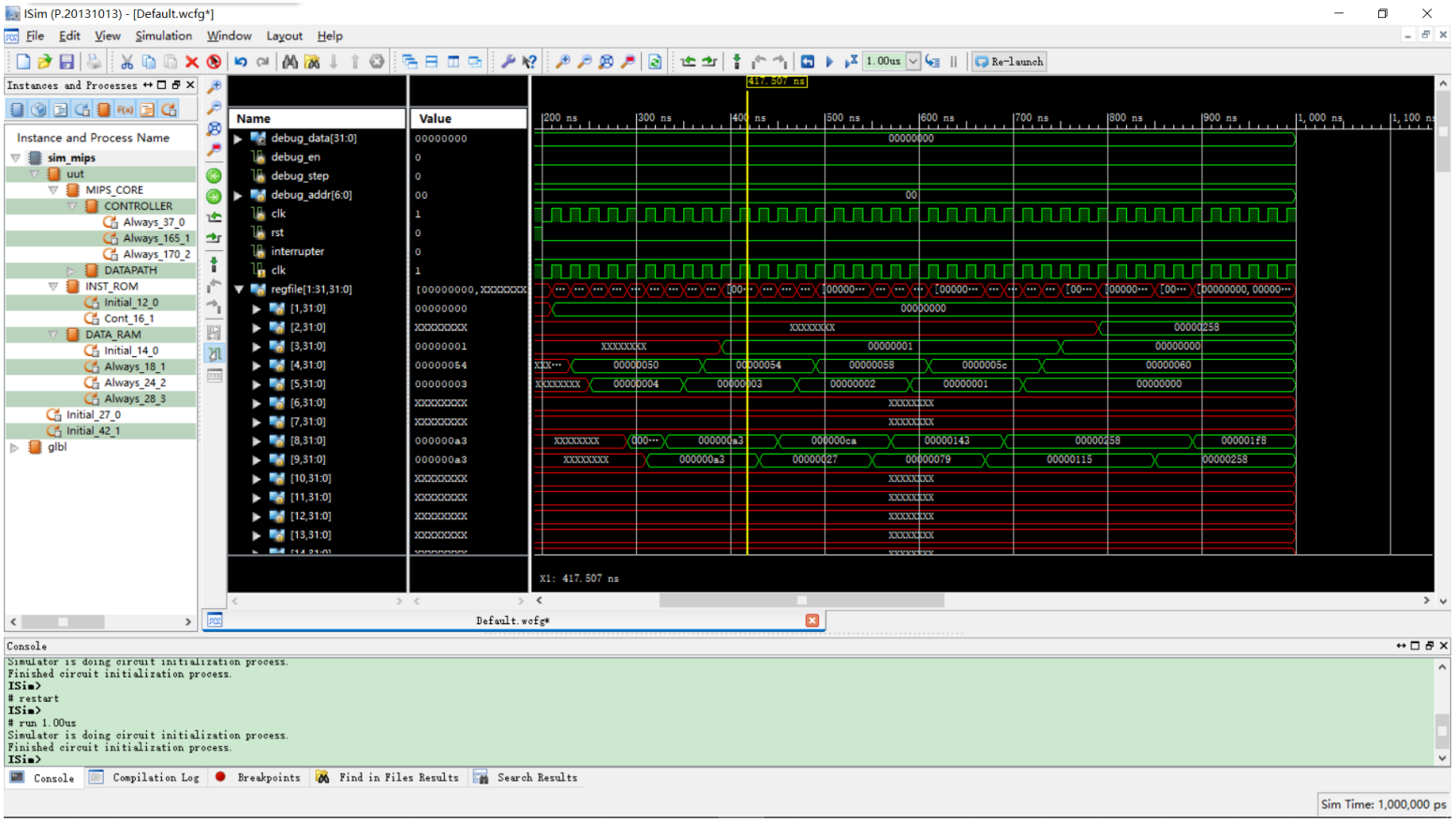
（结尾部分）



regfile 波形（开始部分）



结尾：



在 sword 开发板上的仿真结果（最终结果）：

下文节选一些中间过程（但是因为首先已经跑通到了终点，所以寄存器最后的结果一直在）

五、 讨论与心得

写错代码还是很容易的。